Remote Objects Message Exchange (ROME)

Scott Burleigh
Jet Propulsion Laboratory, California Institute of Technology
Scott.C.Burleigh@jpl.nasa.gov

Problem

The performance of a single program running on a single processor is limited by the character of the processor. Moreover, the cost and difficulty of developing and sustaining programs tend to increase as their size and complexity increase. Clearly there ought to bc some advantage in partitioning powerful application software into relatively small and simple components that can run in parallel on multiple processors: the software should run faster and it should be cheaper and easier to deploy.

Object-oriented software development, using languages such as C++, offers promise as an effective means of partitioning functionality to reduce programming cost, but the problem of parallelization has proven less tractable. Not all parallelization improves performance, because communicate on costs can offset the performance gains achieved through concurrent executi on. In addition, the balance between these two can change in response to changes in operating conditions: changing (e.g., seasonal) usage patterns, changes in network traffic, migration to new hardware, etc. It is therefore important to bc able to change the manner in which elements of an application are distributed, both during development and in practical operation, but historical y such change has been risky and expensive: interprocess communication (I PC) has required specialized skil ls of developers, distributed software can be difficult to test and debug, and support for operation in a heterogeneous platform environment has often been inadequate.

Objective

Remote Obj ects Message Exchange (ROME) [Bur93] is an attempt to provide a single relatively simple, universally available abstraction for data communication among C++ objects, It aims to enable the C++ application developer to specify objects' interactions with other objects wholly in terms of the application domain, without concern for the details of i nterprocess communication. Every ROME-compliant object is conceptual l y a network peer of every other, as if each one were (for example) a separate UNIX process.

Strategy

Some types of technology common] y used to implement distributed object-oriented software were specifically discarded during the design of ROME.

Remote Procedure Calls.

Most C++ programmers are former C programmers, and the standard mechanism for flow of control and data in a C program is the function call. One way to implement C++ inter-object communication would be to enable the invocation of the member functions ("methods") of objects residing on remote machines, as in [WHD89]. However, the commonly understood semantics of functions calls are inherently synchronous: execution is suspended at the point of function invocation, control is passed to the specified function, the function maps input parameters to a unique output value (often with some side effects) and relinquishes control, and execution resumes at the first statement foil owi ng the function call. The possibility of a program doing multiple things concurrently is alien to these semantics.

Naturally, it's possible to provide a mechanism to enable a function's side effects to continue after the function has returned control, but this changes the rules of programming in ways that the developer may not fully understand [CK92]. Furthermore, RPC schemes often rely on the availability of distributed virtual m emery, which Ii r-nits their usefulness to environments with sophisticated operating systems and high available network bandwidth.

IPC objects,

Another approach would be to define classes of objects whose methods can be invoked by "client" code (standalone functions or the methods of other objects) to perform interprocess communication. The drawback to this mechanism is that it would require that application code know which application objects will be local (such that their methods may simply be invoked using the standard C++ call syntax) and which will be remote (accessible only through IPC objects). This conflicts with our goal of m ini mum-effort redistribution of application functionality.

The ROME strategy is instead to provide base classes which encapsulate interobject communication capability in simple asynchronous message exchange methods. ROME-compliant objects are instances of classes that inherit from these base classes; they interact by sending actual messages, not by directly invoking one another's member functions, and they do so whether they reside in the same address space, different address spaces on the same processor, or different processors. The developer of a ROME application works with a single, simple abstraction for all communication among obj ccts that are potential] y di stributed, whether or not they will actually be distributed at run time.

Note that only ROME-compliant objects can send messages; stand-alone C or C++ functions are specifically excluded from participation in ROME. That is, ROME is intended to encourage "pure" object-oriented programming in which all processing is accomplished by objects' methods; a ROME application is a population of cooperating autonomous, quasi-animate objects that exchange messages with one another and, possibly, with other objects in other applications. The rationale for this constraint is partly aesthetic -- a homogeneous population of animate peers seems "cleaner" and simpler than a body of non-object client code that calls the member functions of some other collection of passive objects -- but it has a practical basis as well: absent an extension of the C++ language or a multithreading mechanism (either of whi ch would to some degree conflict with our goal of universal availability), it is much easier to distribute instances of cl asses at run time than instances of functi ons. Peer-to-peer obj ect architect ure seems to be a good fit for parallel processing.

Implementation

Handwritten correspondence being a familiar model for asynchronous communication, ROME is built on a mail service metaphor. Like any postal system, ROME comprises two distinct organizational structures:

Infrastructure

A ROME "universe" comprises one or more *districts*, each roughly corresponding to a local area network; communication delays among ROME objects within a single district should be on the order of seconds or less, while delays in communication bet ween objects in different di stricts may be arbitrarily longer. Each district comprises one or more postal zones, each instantiated as a single operating system task or process. Each zone has a singl e *post office* object, Each post office has one *receiving bay* object for each protocol by which the zone can receive messages, and it also has one *shipping bay* object for every other zone that receives messages by one of the protocols by which this zone knows how to send data. These shipping and receiving bays encapsulate all knowledge of IPC protocols in the application, The post offices of two different zones can communicate if they are either directly mutually reachable (i.e., they share a common protocol) or indirectly mutually reachable (i.e., they can both communicate, either directly or indirectly, with some intermediary zone's post Office).

Utilization

The "customers" of this postal service infrastructure are ROME-compliant application software objects cal led *correspondents,* instances of cl asses derived from the ROME Correspondent base class. Every correspondent resides in a single zone (but some kinds of correspondent can migrate from zone to zone). A collection of correspondents that work together to accomplish some well-defined task is a *community*; typically each instance of a discrete application is likely to be a separate community, A community must be wholly contained within a single district but may comprise objects residing in any number of different zones in that district. Note, though, that correspondents may exchange messages with correspondents in other communit i es. Any correspondent may exchange messages with any other correspondent in any district of the same "universe" provided the post offices of the zones in which the two correspondents reside can communicate,

Each correspondent in a zone is assigned a distinct post office ***box*** into which messages addressed to that correspondent are placed by the post office. The mailing address of any correspondent is the concatenation of the district number and zone number of the zone it resides in, together with its box number.

A ROME application is developed by defining application-specific correspondent class definitions, compiling them, and linking them with ROME libraries and object files. The resulting executable -- which, when loaded, becomes a zone -- automatically constructs a post office, taking self-configuration parameters from command line arguments (district name, zone name, and specifications for all receiving bays). The post office integrates itself into its assigned district and the rest of the universe, and then enters an event loop in which it receives messages from other zones and distributes them to its customers (the correspondents residing in the zone). Correspondents' methods are invoked only by the post office itself, which "calls back" to them when mail arrives that is addressed to them.

In the course of performing a method, a correspondent may send a message to another correspondent; it does so by invoking a post office member function named post. post examines the destination address of the message. If the district/zone of the address is the same as that of the sender (the local zone), then the message is simply appended to a queue of undelivered mail; otherwise the message is handed to appropriate shipping bay for immediate interprocess transmission (see Figure 1). When the correspondent method concludes and the post office's event loop regains control, all undelivered messages are delivered (possibly causing the posting of additional messages, etc.). After no more undelivered mail remains in the post office's queue, the post office waits for a message from another zone to arrive. When one does, the appropriate receiving bay receives it and inserts it into the post office's queue of undelivered mail; this triggers delivery of all undelivered mail, possibly causing the posting of additional messages, and so on,

Features

Projects

ROME is based on asynchronous message exchange, but some programming problems require synchronous processing: some subset of the state of a given object must remain unchanged until the object receives a response to a message it has sent. To accommodate these situations, ROME includes a *project* mechanism. A project is an object that encapsulates a message-handling context. To initiate pseudo-synchronous

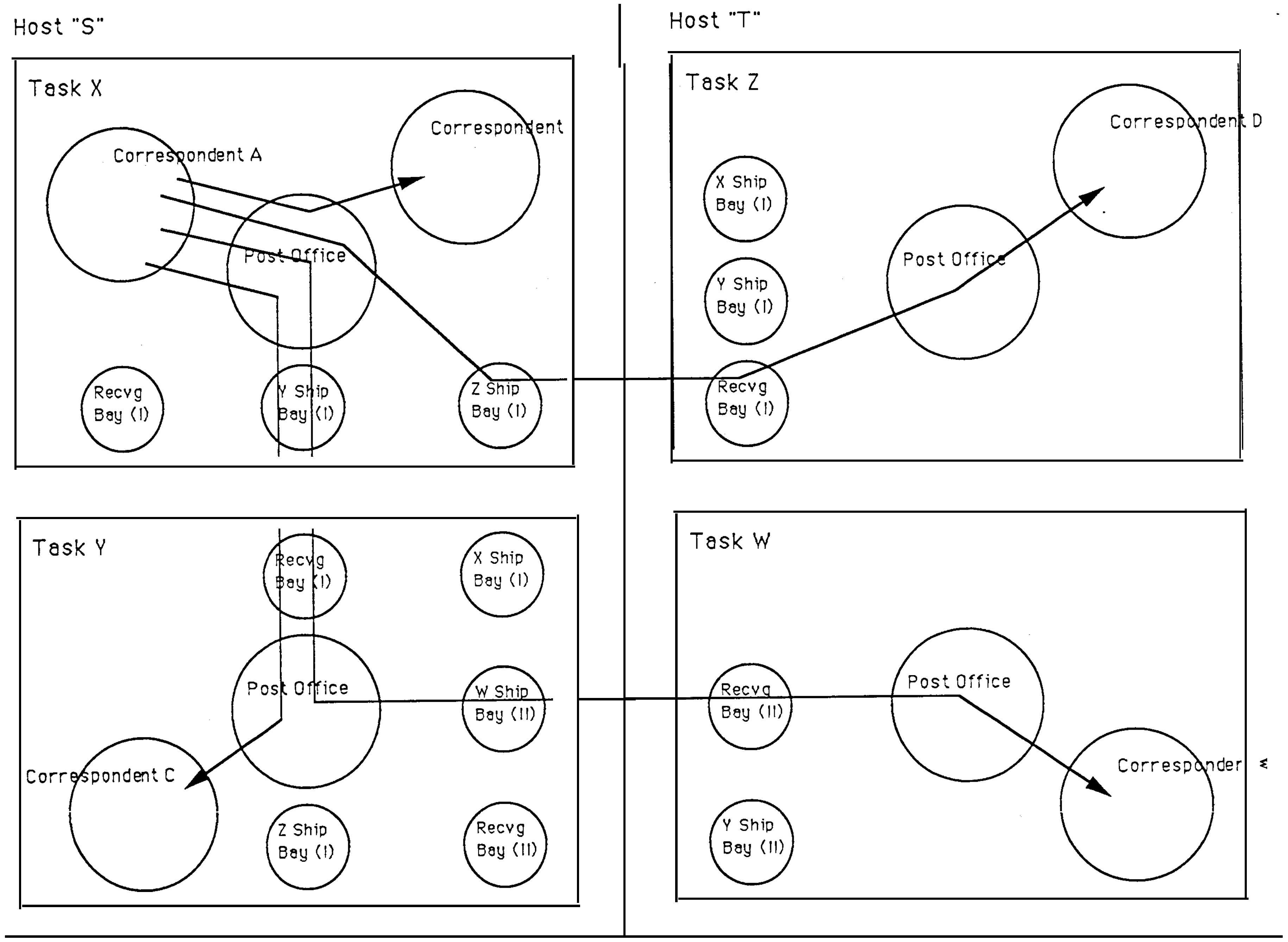

Figure . ROME Message Routing

communication, a correspondent creates a project and attaches that project's ID to the message it sends; it can then return control to the post office. The recipient of the message eventually sends a reply message citing the same project ID. When the post office receives a message citing a project ID it calls back to the handler specified for that project (rather than to the addressee's standard message handler); the handler resumes processing, in the same context in which the correspondent sent the original message.

Timeout intervals can also be specified for proj ects. If a project timeout expires before a reply is received that would cause it to be resumed, the project handler is called anyway and informed that the requested reply is overdue, Moreover, a correspondent can request arbitrary "wake-up calls" at specified intervals by creating a project with a timeout and simply suspending it, without citing it in an issued message.

Note that any number of messages can be handled by the correspondents in a zone, including the initiator of a project, while that project initiator waits for a reply to a message, Each project is, in effect, an extremely lightweight thread of object method execution; that is, projects implement a rudimentary form of platform-independent fine-grained multithreading at the object level.

Referral

The recipient of a message requiring a reply might not be equipped to compute an adequate response. When this is the case, the correspondent can refer (relay) the message to another correspondent for processing. Messages can be re-referred any number of times, but the reply to a referred message always goes directly to the original issuer rather than back through the chain of referees.

Broadcast

Associated with each message is an integer that identifies the *subject* of the message; subjects play the same role as member function names or method selectors in standard object-oriented programming, sel ecti ng the function that is to process the message when it is received. A correspondent may *subscribe* to a subject, and the issuer of a message can ***publish*** it simply by specifying no particular addressee; publishing a message causes copies of it to be sent automatically to all correspondents who have subscribed to the message's subject.

Anomaly handling

ROME provides an anomaly-handling object that encapsulates system and application anomaly processing. User-written anomaly handling can be substituted for the standard methods (which do little more than print messages to stderr) without affecting any ROME or application code.

Migration

As noted above, correspondents derived from the ROME RovingCorrespondent base class can migrate from one zone to another. This may enable support for such capabilities as automatic load balancing in the future.

infrastructure configuration

ROME districts configure themselves automatically as the zones that comprise them are initiated, The post office of the first zone in a district automatically becomes the district's registrar and writes an ASCII string describing itself (including all information needed to communicate with it via its receiving bays) to a "locator" string in a predesignated location. Each subsequent new zone's post office reads every district's locator string and registers with the indicated registrars. Each registrar maintains a list of all active zones in its district, and in response to each registration message it sends a copy of this list to the new post office for use in communication with other zones in the district -- and also notifies all other post offices in the district of the new post office's arrival. If a registrar terminates, the other post offices in the district compete to become the new registrar; each is equipped to do so, since all post offices know about all other post offices.

In the event that an object in zone A sends a message to an object in zone 13, where the post office of B is unable to receive data using any of the protocols that A's post office can transmit on, A's post office will send the message to some intermediary zone C. C will be a zone that can receive what A sends and can in turn send data using either one of B's protocols or one understood by yet another intermediary D, and soon until the message eventual l y reaches B. Routes through these gateways are automat i call y computed by post offices as new post offices register, and routes are automatically recomputed when post offices that served as gateways terminate.

No configuration files, dedicated configuration servers, routing tables, or environment variables need be modified to change the configuration of a ROME communication universe, since none are used.

Platform independence

ROME was designed for minimal dependence on features of specific compilers or operating systems: the library calls it relies on are those that are in most cases available in any good C++ development environment. Platform dependencies are isolated in a single header file. ROME has been successfully and relatively easily ported from SunOS 4.3 on SPARC machines to HP-UX, to IRIX on Silicon Graphics workstations, and to VxWorks on Heurikon MC-68040-based single-board computers.

Historically, a stubborn obstacle to easy distributed computing has been differences in binary data representation on different machines: byte order in integers, alignment in structures, etc. ROME's solution to this problem is *parcels*. A parcel is an instance of a class derived from the Parcel base class, which has two pure virtual functions: **declare** and **acquire**. **declare** marshals the content of a parcel into an array of bytes in a standard format, and **acquire** reconstitutes the parcel's content from such an array. Simple parcel classes corresponding to the C++ built-in types are supplied with ROME, and instances of these classes may in general be used interchangeably with C++ ints, longs, floats, etc. Additional parcel classes for arbitrarily complex application-specific structures are fairly easy to define, provided the attributes of these classes are themselves all instances of parcel classes: a **declare** function for a new parcel class amounts to little more than invoking the **declare** functions of all the attributes of the class.

Whenever a correspondent sends a message, a single (arbitrarily y complex) parcel can be sent as the "content" of the message. Since the Message class is itself derived from Parcel, it's simple for post offices to export complex objects to zones running on foreign computers and be assured that they will be reconstituted correctly when they arrive.

The definitions of the parcel classes for the C++ built-in types are hardware-dependent, but they should be the only code that would need to be rewritten in order to port ROME -- and all ROME applications -- to new hardware.

<u>Protocol independence</u>

All knowledge of specific protocols is encapsulated in shipping and receiving bay objects. At this time ROME supports only TCP/IP and UDP/IP protocols, but adapting all of ROME (and all ROME applications) to -- for example -- invisibly communicate using DCE RPCs should entail writing only a few hundred lines of code.

<u>Support for persistence</u>

Entire zones may be configured either for transient existence (instantiation in heap memory) or, where the Object Store™ object database management system is available, for persistence on magnetic disk. This decision is made at the time a ROME application is compiled, by the selection of a single header file; no application code need be modified.

<u>Support for alternative event contexts</u>

The basic structure of a ROME application is an event loop, in which the post office responds to message-arrival events by calling correspondents' message-handling functions. In some applications, other types of events in addition to ROME message arrivals may need to be handled. Variants of the standard ("std") post office event loop are provided for this purpose: the "ui" event loop handles keyboard keystrokes, the "xui" event loop handles X Windows events (keystrokes and mouse gestures), and the "wtk" event loop continuously recomputes a WorldToolKit™ virtual world while waiting for ROME messages to arrive.

Comparison with CORBA

The design goals of ROME in many way resemble those of the Common Obj ect Request Broker Architecture (CORBA) [OMG91], but the designs themselves differ in several significant ways.

1. Like ROME, CORBA implementations enable objects to pass control and data among themselves without knowing one another's implementation details. Unlike ROME, the intent of CORBA is to support this interaction among objects implemented in different programming 1 anguages; ROME supports only C++ inter-object communicate on at this time.

2. CORBA is built on a client/server RPC communications model; ROME presents itself to the application programmer as a system for message-passing among peer objects.

3. CORBA inter-object communication, being based on RPCS, is naturally synchronous but single-threaded. ROME inter-object communication, being based on message passing, is naturally asynchronous but includes support for multithreaded pseudo-synchronous communicate on (proj ect s).

4. CORBA enforces type discipline on interactions among objects, with type declarations residing in compiled Interface Definition Language (IDL) code or in an Interface Repository; a client must have knowledge of the interface of an object in order to invoke one of its methods. ROME enables any message to be sent to any object of any (correspondent) type; it requires the receiving objects to be responsible for inspecting the subjects of the messages they receive and responding appropriately. This simplifies operations such as broadcasting but precludes compile-time error detection.

In short, the emphasis in the design of CORBA seems to be on support for retaining a familiar function-calling programming style in a distributed environment of platform (including language) heterogeneity, with consequent support for parallel processing. The emphasis in the design of ROME is on support for parallel processing in an event-loop/callback programming style, with consequent support for pl atform (though not language) heterogeneit y.

Acknowledgement

This research was carried out by the Jet Propul sion Laboratory, California Institute of

Technology, under a contract with the National Aeronautics and Space Administration.

References

[Bur93] Scott Burleigh. "ROME: Distributing C++ Object Systems", IEEE Parallel & Distributed Technology Systems & Applications, pp. 21-32, May 1993.

[CK92] K. M. Chandy and C. Kesselman. "CC++: A Declarative Concurrent Object-Oriented Programming Notation", Technical Report Cal Tech-CS-TR-92-01, California institute of Technology, 1992.

[OMG91] Common Object Request Broker Architecture and Specification, OMG Document 91.12,1, Object Management Group, December 1991.

[WHD89] Andrew A. Chien, Waldemar Horwat, and William J. Dally. "Experience with CST: Programming and Implementation", SIGPLAN 89 Conference on Programming Language Design and Implementation, 1989.